Introduction to String Searching Algorithms

Rabin-Karp and Knuth-Morris-Pratt Algorithms

By **TheLlama**– *TopCoder Member*

More Resources

Discuss the article in the forums

The fundamental string searching (matching) problem is defined as follows: given two strings – a text and a pattern, determine whether the pattern appears in the text. The problem is also known as "the needle in a haystack problem."

The "Naive" Method
Its idea is straightforward — for every position in the text, consider it a starting position of the pattern and see if you get a match.

```
function brute_force(text[], pattern[])
{
  // let n be the size of the text and m the size of the
  // pattern

  for(i = 0; i < n; i++) {
    for(j = 0; j < m && i + j < n; j++)
      if(text[i + j] != pattern[j]) break;
      // mismatch found, break the inner loop
    if(j == m) // match found
  }
}
```

The "naive" approach is easy to understand and implement but it can be too slow in some cases. If the length of the text is n and the length of the pattern m, in the worst case it may take as much as (n * m) iterations to complete the task.

It should be noted though, that for most practical purposes, which deal with texts based on human languages, this approach is much faster since the inner loop usually quickly finds a mismatch and breaks. A problem arises when we are faced with different kinds of "texts," such as the genetic code.

Rabin-Karp Algorithm (RK)
This is actually the “naive” approach augmented with a powerful programming technique – the hash function.

Every string s[] of length m can be seen as a number H written in a positional numeral system in base B (B >= size of the alphabet used in the string):

$$H = s[0] * B^{(m-1)} + s[1] * B^{(m-2)} + \ldots + s[m-2] * B^{1} + s[m-1] * B^{0}$$

If we calculate the number H (the hash value) for the pattern and the same number for every substring of length m of the text than the inner loop of the “naive” method will disappear – instead of comparing two strings character by character we will have just to compare two integers.

A problem arises when m and B are big enough and the number H becomes too large to fit into the standard integer types. To overcome this, instead of the number H itself we use its remainder when divided by some other number M. To get the remainder we do not have to calculate H. Applying the basic rules of modular arithmetic to the above expression:

$$A + B = C \Rightarrow (A \% M + B \% M) \% M = C \% M$$
$$A * B = C \Rightarrow ((A \% M) * (B \% M)) \% M = C \% M$$

We get:

$$H \% M = (((s[0] \% M) * (B^{(m-1)} \% M)) \% M + ((s[1] \% M) * (B^{(m-2)} \% M)) \% M + \ldots$$
$$\ldots + ((s[m-2] \% M) * (B^{1} \% M)) \% M + ((s[m-1] \% M) * (B^{0} \% M)) \% M) \% M$$

The drawback of using remainders is that it may turn out that two different strings map to the same number (it is called a collision). This is less likely to happen if M is sufficiently large and B and M are prime numbers. Still this does not allow us to entirely skip the inner loop of the “naive” method. However, its usage is significantly limited. We have to compare the “candidate” substring of the text with the pattern character by character only when their hash values are equal.

Obviously the approach described so far would be absolutely useless if we were not able to calculate the hash

value for every substring of length m in the text in just one pass through the entire text. At first glance to do these calculations we will again need two nested loops: an outer one – to iterate through all possible starting positions — and an inner one – to calculate the hash function for every starting position. Fortunately, this is not the case. Let's consider a string s[], and let's suppose we are to calculate the hash value for every substring in s[] with length say m = 3. It is easy to see that:

$$\mathbf{H_0 = H_{s[0]...s[2]} = s[0] * B^2 + s[1] * B + s[2]}$$

$$\mathbf{H_1 = H_{s[1]..s[3]} = s[1] * B^2 + s[2] * B + s[3]}$$

$$\mathbf{H_1 = (H_0 - s[0] * B^2) * B + s[3]}$$

In general:

$$\mathbf{H_i = (H_{i-1} - s[i- 1] * B^{m-1}) * B + s[i + m - 1]}$$

Applying again the rules of modular arithmetic, we get:

$$\mathbf{H_i \% M = ((((H_{i-1} \% M - ((s[i- 1] \% M) * (B^{m-1} \% M)) \% M) \% M) * (B \% M)) \% M + s[i + m - 1] \% M) \% M}$$

Obviously the value of $(H_{i-1} - s[i - 1] * B^{m-1})$ may be negative. Again, the rules of modular arithmetic come into play:

$$\mathbf{A - B = C => (A \% M - B \% M + k * M) \% M = C \% M}$$

Since the absolute value of $(H_{i-1} - s[i - 1] * B^{m-1})$ is between 0 and (M – 1), we can safely use a value of 1 for k.

Pseudocode for RK follows:

```
// correctly calculates a mod b even if a < 0
```

```
function int_mod(int a, int b)
{
  return (a % b + b) % b;
}

function Rabin_Karp(text[], pattern[])
{
  // let n be the size of the text, m the size of the
  // pattern, B - the base of the numeral system,
  // and M - a big enough prime number

  if(n < m) return; // no match is possible

  // calculate the hash value of the pattern
  hp = 0;
  for(i = 0; i < m; i++)
    hp = int_mod(hp * B + pattern[i], M);

  // calculate the hash value of the first segment
  // of the text of length m
  ht = 0;
  for(i = 0; i < m; i++)
    ht = int_mod(ht * B + text[i], M);

  if(ht == hp) check character by character if the first
               segment of the text matches the pattern;

  // start the "rolling hash" - for every next character in
  // the text calculate the hash value of the new segment
  // of length m; E = (Bm-1) modulo M
```

```
  for(i = m; i < n; i++) {
    ht = int_mod(ht - int_mod(text[i - m] * E, M), M);
    ht = int_mod(ht * B, M);
    ht = int_mod(ht + text[i], M);

    if(ht == hp) check character by character if the
                 current segment of the text matches
                 the pattern;
  }
}
```

Unfortunately, there are still cases when we will have to run the entire inner loop of the “naive” method for every starting position in the text – for example, when searching for the pattern “aaa” in the string “aaaaaaaaaaaaaaaaaaaaaaaa” — so in the worst case we will still need (n * m) iterations. How do we overcome this?

Let’s go back to the basic idea of the method — to replace the string comparison character by character by a comparison of two integers. In order to keep those integers small enough we have to use modular arithmetic. This causes a “side effect” — the mapping between strings and integers ceases to be unique. So now whenever the two integers are equal we still have to “confirm” that the two strings are identical by running character-by-character comparison. It can become a kind of vicious circle…

The way to solve this problem is “rational gambling,” or the so called “double hash” technique. We “gamble” — whenever the hash values of two strings are equal, we assume that the strings are identical, and do not compare them character by character. To make the likelihood of a “mistake” negligibly small we compute for every string not one but two independent hash values based on different numbers B and M. If both are equal, we assume that the strings are identical. Sometimes even a “triple hash” is used, but this is rarely justifiable from a practical point of view.

The “pure” form of “the needle in a haystack problem” is considered too straightforward and is rarely seen in

programming challenges. However, the “rolling hash” technique used in RK is an important weapon. It is especially useful in problems where we have to look at all substrings of fixed length of a given text. An example is “the longest common substring problem”: given two strings find the longest string that is a substring of both. In this case, the combination of binary search (BS) and “rolling hash” works quite well. The important point that allows us to use BS is the fact that if the given strings have a common substring of length n, they also have at least one common substring of any length m < n. And if the two strings do not have a common substring of length n they do not have a common substring of any length m > n. So all we need is to run a BS on the length of the string we are looking for. For every substring of the first string of the length fixed in the BS we insert it in a hash table using one hash value as an index and a second hash value (“double hash”) is inserted in the table. For every substring of the fixed length of the second string, we calculate the corresponding two hash values and check in the table to see if they have been already seen in the first string. A hash table based on open addressing is very suitable for this task.

Of course in “real life” (real challenges) the number of the given strings may be greater than two, and the longest substring we are looking for should not necessarily be present in all the given strings. This does not change the general approach.

Another type of problems where the “rolling hash” technique is the key to the solution are those that ask us to find the most frequent substring of a fixed length in a given text. Since the length is already fixed we do not need any BS. We just use a hash table and keep track of the frequencies.

Knuth-Morris-Pratt Algorithm (KMP)

In some sense, the “naive” method and its extension RK reflect the standard approach of human logic to “the needle in a haystack problem”. The basic idea behind KMP is a bit different. Let’s suppose that we are able, after one pass through the text, to identify all positions where an existing match with the pattern ends. Obviously, this will solve our problem. Since we know the length of the pattern, we can easily identify the starting position of every match.

Is this approach feasible? It turns out that it is, when we apply the concept of the automaton. We can think of an automaton as of a kind of abstract object, which can be in a finite number of states. At each step some information is presented to it. Depending on this information and its current state the automaton goes to a

new state, uniquely determined by a set of internal rules. One of the states is considered as “final”. Every time we reach this “final” state we have found an end position of a match.

The automaton used in KMP is just an array of “pointers” (which represents the “internal rules”) and a separate “external” pointer to some index of that array (which represents the “current state”). When the next character from the text is presented to the automaton, the position of the “external” pointer changes according to the incoming character, the current position, and the set of “rules” contained in the array. Eventually a “final” state is reached and we can declare that we have found a match.

The general idea behind the automaton is relatively simple. Let us consider the string

A B A B A C

as a pattern, and let’s list all its prefixes:

0 /the empty string/
1 A
2 A B
3 A B A
4 A B A B
5 A B A B A
6 A B A B A C

Let us now consider for each such listed string (prefix) the longest proper suffix (a suffix different from the string itself), which is at the same time a prefix of it:

0 /the empty string/
1 /the empty string/
2 /the empty string/
3 A
4 A B

5 A B A

6 /the empty string/

It's easy to see that if we have at some point a partial match up to say the prefix (A B A B A) we also have a partial match up to the prefixes (A B A), and (A) – which are both prefixes of the initial string and suffix/prefixes of the current match. Depending on the next "incoming" character from the text, three cases arise:

1. The next character is C. We can "expand" the match at the level of the prefix (A B A B A). In this particular case this leads to a full match and we just notice this fact.

2. The next character is B. The partial match for the prefix (A B A B A) cannot be "expanded". The best we can do is to return to the largest different partial match we have so far – the prefix (A B A) and try to "expand" it. Now B "fits" so we continue with the next character from the text and our current "best" partial match will become the string (A B A B) from our "list of prefixes".

3. The "incoming" character is, for example, D. The "journey" back to (A B A) is obviously insufficient to "expand" the match. In this case we have to go further back to the second largest partial match (the second largest proper suffix of the initial match that is at the same time a prefix of it) – that is (A) and finally to the empty string (the third largest proper suffix in our case). Since it turns out that there is no way to "expand" even the empty string using the character D, we skip D and go to the next character from the text. But now our "best" partial match so far will be the empty string.

In order to build the KMP automaton (or the so called KMP "failure function") we have to initialize an integer array F[]. The indexes (from 0 to m – the length of the pattern) represent the numbers under which the consecutive prefixes of the pattern are listed in our "list of prefixes" above. Under each index is a "pointer" – that identifies the index of the longest proper suffix, which is at the same time a prefix of the given string (or in other words F[i] is the index of next best partial match for the string under index i). In our case (the string A B A B A C) the array F[] will look as follows:

F[0] = 0

F[1] = 0
F[2] = 0
F[3] = 1
F[4] = 2
F[5] = 3
F[6] = 0

Notice that after initialization F[i] contains information not only about the largest next partial match for the string under index i but also about every partial match of it. F[i] is the first best partial match, F[F[i]] – is the second best, F[F[F[i]]] – the third, and so on. Using this information we can calculate F[i] if we know the values F[k] for all k < i. The best next partial match of string i will be the largest partial match of string i – 1 whose character that "expands" it is equal to the last character of string i. So all we need to do is to check every partial match of string i – 1 in descending order of length and see if the last character of string i "expands" the match at this level. If no partial match can be "expanded" than F[i] is the empty string. Otherwise F[i] is the largest "expanded" partial match (after its "expansion").

In terms of pseudocode the initialization of the array F[] (the "failure function") may look like this:

```
// Pay attention!
// the prefix under index i in the table above is
// is the string from pattern[0] to pattern[i - 1]
// inclusive, so the last character of the string under
// index i is pattern[i - 1]

function build_failure_function(pattern[])
{
  // let m be the length of the pattern

  F[0] = F[1] = 0; // always true
```

```
  for(i = 2; i <= m; i++) {
    // j is the index of the largest next partial match
    // (the largest suffix/prefix) of the string under
    // index i - 1
    j = F[i - 1];
    for( ; ; ) {
      // check to see if the last character of string i -
      // - pattern[i - 1] "expands" the current "candidate"
      // best partial match - the prefix under index j
      if(pattern[j] == pattern[i - 1]) {
        F[i] = j + 1; break;
      }
      // if we cannot "expand" even the empty string
      if(j == 0) { F[i] = 0; break; }
      // else go to the next best "candidate" partial match
      j = F[j];
    }
  }
}
```

The automaton consists of the initialized array F[] (“internal rules”) and a pointer to the index of the prefix of the pattern that is the best (largest) partial match that ends at the current position in the text (“current state”). The use of the automaton is almost identical to what we did in order to build the “failure function”. We take the next character from the text and try to “expand” the current partial match. If we fail, we go to the next best partial match of the current partial match and so on. According to the index where this procedure leads us, the “current state” of the automaton is changed. If we are unable to “expand” even the empty string we just skip this character, go to the next one in the text, and the “current state” becomes zero.

```
function Knuth_Morris_Pratt(text[], pattern[])
{
  // let n be the size of the text, m the
  // size of the pattern, and F[] - the
  // "failure function"

  build_failure_function(pattern[]);

  i = 0; // the initial state of the automaton is
         // the empty string

  j = 0; // the first character of the text

  for( ; ; ) {
    if(j == n) break; // we reached the end of the text

    // if the current character of the text "expands" the
    // current match
    if(text[j] == pattern[i]) {
      i++; // change the state of the automaton
      j++; // get the next character from the text
      if(i == m) // match found
    }

    // if the current state is not zero (we have not
    // reached the empty string yet) we try to
    // "expand" the next best (largest) match
    else if(i > 0) i = F[i];
```

```
    // if we reached the empty string and failed to
    // "expand" even it; we go to the next
    // character from the text, the state of the
    // automaton remains zero
    else j++;
  }
}
```

Many problems in programming challenges focus more on the properties of KMP's "failure function," rather than on its use for string matching. An example is: given a string (a quite long one), find all its proper suffixes that are also prefixes of it. All we have to do is just to calculate the "failure function" of the given string and using the information stored in it to print the answer.

A typical problem seen quite often is: given a string find its shortest substring, such that the concatenation of one or more copies of it results in the original string. Again the problem can be reduced to the properties of the failure function. Let's consider the string

A B A B A B

and all its proper suffix/prefixes in descending order:

1 A B A B
2 A B
3 /the empty string/

Every such suffix/prefix uniquely defines a string, which after being "inserted" in front of the given suffix/prefix gives the initial string. In our case:

1 A B
2 A B A B
3 A B A B A B

Every such “augmenting” string is a potential “candidate” for a string, the concatenation of several copies of which results in the initial string. This follows from the fact that it is not only a prefix of the initial string but also a prefix of the suffix/prefix it “augments”. But that means that now the suffix/prefix contains at least two copies of the “augmenting” string as a prefix (since it’s also a prefix of the initial string) and so on. Of course if the suffix/prefix under question is long enough. In other words, the length of a successful “candidate” must divide with no remainder the length of the initial string.

So all we have to do in order to solve the given problem is to iterate through all proper suffixes/prefixes of the initial string in descending order. This is just what the “failure function” is designed for. We iterate until we find an “augmenting” string of the desired length (its length divides with no remainder the length of the initial string) or get to the empty string, in which case the “augmenting” string that meets the above requirement will be the initial string itself.

Rabin-Karp and Knuth-Morris-Pratt at TopCoder

In the problem types mentioned above, we are dealing with relatively “pure” forms of RK, KMP and the techniques that are the essence of these algorithms. While you’re unlikely to encounter these pure situations in a TopCoder SRM, the drive towards ever more challenging TopCoder problems can lead to situations where these algorithms appear as one level in complex, “multilayer” problems. The specific input size limitations favor this trend, since we will not be presented as input with multimillion character strings, but rather with a “generator”, which may be by itself algorithmic in nature. A good example is “InfiniteSoup,” Division 1 – Level Three, SRM 286.

Topcoder is also on